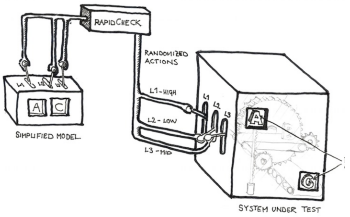
Common Challenges in Unit Testing

▼ Mocking in Unit Tests

What is Mocking?

- Mocking is a testing technique where you replace real objects with simulated ones (known as mocks) to isolate the unit of code being tested.
- This helps simulate certain behaviors or conditions that might be difficult or impractical to replicate in real environments, such as database connections, network calls, or third-party APIs.

Generating test cases so you don't have to - Spotify Engineering



https://engineering.atspotify.com/2015/06/rapid-check/

Why Mocking?

- Mocking helps in testing components in isolation by removing external dependencies.
- Mocks are faster than interacting with actual databases or services.
- They ensure that tests are repeatable with predictable outcomes.

Types of Mocks

1. `Mocks`: Simulate behaviour by returning predefined outputs for method calls.
2. `Stubs`: Simple objects that return pre-defined responses to method calls.
3. `Spies`: Track the behaviour of objects, including method invocations.
4. `Fakes`: Real working implementations of interfaces or services but simplified for testing purposes.

Example of Mocking in Java

Using `Mockito`, a popular mocking framework for Java:

```
import static org.mockito.Mockito.*;
import org.junit.jupiter.api.Test;
import java.util.List;

class MockTest {
    @Test
    void testMock() {
        // Create a mock object of List
        List<String> mockedList = mock(List.class);

        // Define behavior of the mock object
        when(mockedList.get(0)).thenReturn("Hello");

        // Test the behavior
        assertEquals("Hello", mockedList.get(0));

        // Verify that get(0) was called once
        verify(mockedList).get(0);
    }
}
```

In this example, we mock a `List` object, specify a return value for `get(0)`, and verify its behaviour. The actual `List` implementation isn't called, so this is faster and isolates the test from the real object.

When to Avoid Over-Mocking

- Critical parts of the application logic should not be mocked. Testing logic only through mocked objects can mask important issues.
- Mocking too many objects can make tests harder to read and maintain.
- Excessive mocking can result in tests that pass but don’t reflect real-world behaviour.

Why Over-Mocking is Problematic

- **Missed Bugs →** Excessive mocking leads to tests passing despite serious bugs in the real objects.
- **Brittle Tests →** When too many interactions are mocked, small changes in implementation details might break tests unnecessarily.

Example of Over-Mocking

Suppose you are testing a service that depends on a repository and an external API:

```
class MovieServiceTest {
    @Test
    void testMovieService() {
        // Mock repository and API client
        MovieRepository movieRepository = mock(MovieRepository.cl
        ExternalApiClient apiClient = mock(ExternalApiClient.clas

        // Mock behavior
        when(movieRepository.findById(1L)).thenReturn(Optional.of
        when(apiClient.getRating("Inception")).thenReturn(8.8);

        // Test service method
        MovieService movieService = new MovieService(movieReposi
        MovieDetails movieDetails = movieService.getMovieDetails
```

```
        assertEquals(8.8, movieDetails.getRating());
    }
}
```

In this test, the repository and API client are both mocked. If both components are mocked too much, bugs in the real API integration or repository logic may never be caught because tests don't exercise the actual behavior of those components.

▼ Flaky Test in Unit Test

What is a Flaky Test?

- A **flaky test** is a test that passes or fails intermittently without changes to the underlying code.
- This inconsistency makes it difficult to trust test results, leading to confusion and frustration for developers.

Flaky Tests at Google and How We Mitigate Them

by John Micco At Google, we run a very large corpus of tests continuously to validate our code submissions. Everyone from developers to pr...

https://testing.googleblog.com/2016/05/flaky-tests-at-google-and-how-we.html

Why Do Flaky Tests Occur?

💡

- Relying on databases, APIs, or services that are not under your control can introduce randomness or latency into tests.
- Tests that depend on specific execution times or timeouts may behave inconsistently.
- Tests involving random data, asynchronous processes, or multi-threading can result in unpredictable outcomes.

Example of a Flaky Test

Consider a unit test that tests a method relying on a timer or random number generator:

```
@Test
public void testRandomNumberGenerator() {
    RandomNumberGenerator generator = new RandomNumberGenerato
```

```
r();

    // Test expects the number to always be 5
    assertEquals(5, generator.generateRandomNumber());
}
```

If the `generateRandomNumber()` method generates random values each time, this test will fail intermittently, making it flaky.

Why Flaky Tests Are Problematic

- **Developer Time Wasted →** Developers spend excessive time trying to diagnose whether a failure is due to a legitimate bug or a flaky test.
- **CI/CD Pipeline Failures →** Flaky tests can cause entire pipelines to fail, delaying code deployments and slowing down the development cycle.
- **Reduced Trust →** Over time, flaky tests erode developer confidence in the test suite, leading to skipped or ignored tests.

How to Mitigate Flaky Tests

1. Use mocks or stubs for external services and APIs, ensuring tests do not rely on factors outside of your control.
2. For tests dependent on timers, use fixed time references (e.g., `System.currentTimeMillis()` mocked to return a constant value).
3. Add retry mechanisms or longer wait times to handle tests that depend on asynchronous processes or event-driven architectures.

Example of Fixing a Flaky Test

For a flaky test that relies on time, mock the current time instead of using real-time references:

```
@Test
public void testTimeDependentFeature() {
    // Use a fixed date/time instead of the real system clock
    Clock fixedClock = Clock.fixed(Instant.parse("2023-09-20T00:(
    MyService service = new MyService(fixedClock);

    // Test service behavior at the fixed time
```

```
    assertTrue(service.isTimeWithinWorkingHours());
}
```

Best Practices for Avoiding Over-Mocking and Flaky Tests

1. **Write Small, Focused Tests**:
   - Each unit test should focus on a single behavior, keeping mocks to a minimum.
2. **Mock Only External Dependencies**:
   - Don’t mock internal logic or core components of your application.
3. **Ensure Deterministic Tests**:
   - Tests should always produce the same result, regardless of environment or timing.
4. **Test Edge Cases**:
   - Include edge cases to ensure the test is robust and catches issues early.